IBM Hight Performance Computing Toolkit
MPI Tracing/Profiling User Manual

Advanced Computing Technology Center
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598

April 4, 2008

Contents

1 Overview **2**

2 System and Software Requirement **2**

3 Compiling and Linking **2**
3.1 AIX on Power . . . 3
3.2 Linux on Power . . . 3
3.3 Blue Gene/L . . . 3
3.4 Blue Gene/P . . . 4

4 Environment Variable **4**

5 Output **7**
5.1 Plain Text File . . . 7
5.2 Viz File . . . 10
5.3 Trace File . . . 10

6 Configuration **10**
6.1 Configuration Function . . . 11
6.2 Data Structure . . . 11
6.3 Utility Functions . . . 13
6.4 Example . . . 15

7 Final Note **16**
7.1 Overhead . . . 16
7.2 Multi-Threading . . . 17

8 Contacts **17**

1 Overview

This is the documentation for the IBM High Performance Computing Toolkit MPI Profiling/Tracing library. This library collects profiling and tracing data for MPI programs.

The library file names and their usage are shown in Table 1.

Name	Usage
libmpitrace.a	library file for both C and Fortran application
mpt.h	C header file

Table 1: Library file names and usage

Note:

1. The C header file is used when it needs to configure the library. Please see Section 6 for details.

2 System and Software Requirement

Current supported architecture/OS and required software are:

- AIX on Power (32 bit and 64 bit)
  - IBM Parallel Environment (PE) for AIX program product and its Parallel Operating Environment (POE).
- Linux on Power (32 bit and 64 bit)
  - IBM Parallel Environment (PE) for Linux program product and its Parallel Operating Environment (POE).
- Blue Gene/L
  - System Software version 3.
- Blue Gene/P

3 Compiling and Linking

The trace library uses the debugging information stored within the binary to map the performance information back to the source code. To use the library, the application must be compiled with the "-g" option.

The application may consider turning off or having lower level of optimization (-O2, -O1,...) when linking with the MPI profiler/tracer. High level optimization will affect the correctness of the debugging information. It may also affect the call stack behavior.

To link the application with the library, add three options to your command line: the option -L/path/to/libraries, where /path/to/libraries is the path where the libraries are located, the option -lmpitrace (this trace library should be before the MPI library -lmpich) in the linking order, and the option -llicense to link the license library. For some platforms, if the shared library liblicense.so is used, you may need to set the environment variable LD_LIBRARY_PATH to $IHPCT_BASE/lib(lib64) to make sure the application finds the correct library during runtime.

3.1 AIX on Power

- C example

```
CC = /usr/lpp/ppe.poe/bin/mpcc_r
TRACE_LIB = -L</path/to/libmpitrace.a> -lmpitrace -llicense

mpitrace.ppe: mpi_test.c
        $(CC) -g -o $@ $< $(TRACE_LIB) -lm
```

- Fortran example

```
FC = /usr/lpp/ppe.poe/bin/mpxlf_r
TRACE_LIB = -L</path/to/libmpitrace.a> -lmpitrace -llicense

swim.ppe: swim.f
        $(FC) -g -o $@ $< $(TRACELIB)
```

3.2 Linux on Power

- C example

```
CC = /opt/ibmhpc/ppe.poe/bin/mpcc
TRACE_LIB = -L</path/to/libmpitrace.a> -lmpitrace -llicense

mpitrace: mpi_test.c
        $(CC) -g -o $@ $< $(TRACE_LIB) -lm
```

- Fortran example

```
FC = /opt/ibmhpc/ppe.poe/bin/mpfort
TRACE_LIB = -L</path/to/libmpitrace.a> -lmpitrace -llicense

statusesf_trace: statusesf.f
        $(FC) -g -o $@ $< $(TRACE_LIB)
```

3.3 Blue Gene/L

- C example

```
BGL_INSTALL = /bgl/BlueLight/ppcfloor
LIBS_RTS = -lrts.rts -ldevices.rts
LIBS_MPI = -L$(BGL_INSTALL)/bglsys/lib -lmpich.rts -lmsglayer.rts $(LIBS_RTS)
XLC_TRACE_LIB = -L</path/to/libmpitrace.a> -lmpitrace -llicense
XLC_RTS = blrts_xlc
XLC_CFLAGS = -I$(BGL_INSTALL)/bglsys/include -g -O -qarch=440 -qtune=440 -qhot

mpitrace_xlc.rts: mpi_test.c
$(XLC_RTS) -o $@ $< $(XLC_CFLAGS) $(XLC_TRACE_LIB) $(LIBS_MPI) -lm
```

- Fortran example

```
BGL_INSTALL = /bgl/BlueLight/ppcfloor
LIBS_RTS = -lrts.rts -ldevices.rts
LIBS_MPI = -L$(BGL_INSTALL)/bglsys/lib -lmpich.rts -lmsglayer.rts $(LIBS_RTS)
TRACE_LIB = -L</path/to/libmpitrace.a> -lmpitrace -llicense
BG_XLF = blrts_xlf
FC_FLAGS = -I$(BGL_INSTALL)/bglsys/include -g -O

statusesf_trace.rts: statusesf.f
        $(BG_XLF) -o $@ $< $(FC_FLAGS) $(TRACE_LIB) $(MPI_LIBS)
```

3.4 Blue Gene/P

- C example

```
BGPHOME=/bgsys/drivers/ppcfloor
CC=$(BGPHOME)/comm/bin/mpicc
CFLAGS = -I$(BGPHOME)/comm/include -g -O
TRACE_LIB = -L</path/to/libmpitrace.a> -lmpitrace -llicense
LIB1 = -L$(BGPHOME)/comm/lib -lmpich.cnk -ldcmfcoll.cnk -ldcmf.cnk
LIB2 = -L$(BGPHOME)/runtime/SPI -lSPI.cna -lpthread -lrt
LIB3 = -lgfortranbegin -lgfortran # please read the NOTE

mpitrace: mpi_test.c
        $(CC) -o $@ $< $(CFLAGS) $(TRACE_LIB) $(LIB1) $(LIB2) -lm
```

NOTE: the C example uses mpicc which currently is based on GNU compiler. In order to accomodate part of the tracing/profiling library that is written in Fortran. It is necessary to link those two GNU Fortran libraries.

- Fortran example

```
BGPHOME=/bgsys/drivers/ppcfloor
CC=$(BGPHOME)/comm/bin/mpif77
FFLAGS = -I$(BGPHOME)/comm/include -g -O
TRACE_LIB = -L</path/to/libmpitrace.a> -lmpitrace -llicense
LIB1 = -L$(BGPHOME)/comm/lib -lmpich.cnk -ldcmfcoll.cnk -ldcmf.cnk
LIB2 = -L$(BGPHOME)/runtime/SPI -lSPI.cna -lpthread -lrt

statusesf: statusesf.f
        $(CC) -o $@ $< $(FFLAGS) $(TRACE_LIB) $(LIB1) $(LIB2)
```

4 Environment Variable

- TRACE_ALL_EVENTS
  The wrappers can be used in two modes. The default value is set to yes and will collect both a timing summary and a time-history of MPI calls suitable for graphical display.

  If this environment variable is set to be yes, it will save a record of all MPI events [1] after MPI_Init(), until the application completes, or until the trace buffer is full.

[1] By default, for MPI ranks 0-255, or for all MPI ranks if there are 256 or fewer processes in MPI_COMM_WORLD. You can change this by setting the TRACE_ALL_TASKS or using configuration described in Section 6.

Another method is to control time-history measurement within the application by calling routines to start/stop tracing:

- Fortran syntax

```
call mt_trace_start()
do work + mpi ...
call mt_trace_stop()
```

- C syntax

```
void MT_trace_start(void);
void MT_trace_stop(void);

MT_trace_start();
do work + mpi ...
MT_trace_stop();
```

- C++ syntax

```
extern "C" void MT_trace_start(void);
extern "C" void MT_trace_stop(void);

MT_trace_start();
do work + mpi ...
MT_trace_stop();
```

To use this control method, the environment variable needs to be disabled (otherwise it would trace all events):

```
export TRACE_ALL_EVENTS=no (bash)
setenv TRACE_ALL_EVENTS no (csh)
```

- TRACE_ALL_TASKS

  When saving MPI event records, it is easy to generate trace files that are just too large to visualize. To cut down on the data volume, the default behavior when you set TRACE_ALL_EVENTS=yes is to save event records from MPI tasks 0-255, or for all MPI processes if there are 256 or fewer processes in MPI_COMM_WORLD. That should be enough to provide a good visual record of the communication pattern. If you want to save data from all tasks, you have to set this environment variable to yes:

```
export TRACE_ALL_TASKS=yes (bash)
setenv TRACE_ALL_TASKS yes (csh)
```

- TRACE_MAX_RANK

To provide more control, you can set MAX_TRACE_RANK=#. For example, if you set MAX_TRACE_RANK=2048, you will get trace data from 2048 tasks, 0-2047, provided you actually have at least 2048 tasks in your job. By using the time-stamped trace feature selectively, both in time (trace_start/ trace_stop), and by MPI rank, you can get good insight into the MPI performance of very large complex parallel applications.

- OUTPUT_ALL_RANKS

  For scalability reason, by default only four ranks will generate plain text files and the events in the trace: rank 0, rank with (min,med,max) MPI communication time. if rank 0 is one of the ranks with (min,med,max) MPI communication time, only three ranks will generate plain text files and events in the trace. If plain text files and events should be output from all ranks, set this environment variable to yes:

  ```
  export OUTPUT_ALL_RANKS=yes (bash)
  setenv OUTPUT_ALL_RANKS yes (csh)
  ```

- TRACEBACK_LEVEL

  In some cases there may be deeply nested layers on top of MPI, and you may need to profile higher up the call chain (functions in the call stack). You can do this by setting this environment variable (default value is 0). For example, setting TRACEBACK_LEVEL=1 tells the library to save addresses starting not with the location of the MPI call (level = 0), but from the parent in the call chain (level = 1).

- SWAP_BYTES

  The event trace file is binary, and so it is sensitive to byte order. For example, Blue Gene/L is big endian, and your visualization workstation is probably little endian (e.g., x86). The trace files are written in little endian format by default. If you use a big endian system for graphical display (examples are Apple OS/X, AIX p-series workstations, etc.), you can set an environment variable

  ```
  export SWAP_BYTES=no (bash)
  setenv SWAP_BYTES no (csh)
  ```

  when you run your job. This will result in a trace file in big endian format.

- TRACE_SEND_PATTERN (Blue Gene/L and Blue Gene/P Only)

  In either profiling or tracing mode there is an option to collect information about the number of hops for point-to-point communication on the torus. This feature can be enabled by setting an environment variable:

  ```
  export TRACE_SEND_PATTERN=yes
  setenv TRACE_SEND_PATTERN yes
  ```

When this variable is set, the wrappers keep track of how many bytes are sent to each task, and a binary file "send_bytes.matrix" is written during MPI_Finalize which lists how many bytes were sent from each task to all other tasks. The format of the binary file is:

$$D_{00}, D_{01}, ...D_{0n}, D_{10}, ..., D_{ij}, ..., D_{nn}$$

where the data type D_{ij} is double (in C) and it represents the size of MPI data sent from rank i to rank j. This matrix can be used as input to external utilities that can generate efficient mappings of MPI tasks onto torus coordinates. The wrappers also provide the average number of hops for all flavors of MPI_Send. The wrappers do not track the message-traffic patterns in collective calls, such as MPI_Alltoall. Only point-to-point send operations are tracked.

The $AverageHops$ for all communications on a given processor is measured as follows:

$$AverageHops = \frac{\sum_i Hops_i \times Bytes_i}{\sum_i Bytes_i}$$

where $Hops_i$ is the distance between the processors for ith MPI communication and $Bytes_i$ is the size of the data transferred in this communication. The logical concept behind this performance metric is to measure how far each byte has to travel for the communication (in average). If the communication processor pair is close to each other in the coordinate, the $AverageHops$ value will tend to be small.

5 Output

After building the binary executable and setting the environment, run the application as you normally would. To have better control for the performance data collected and output, please refer to Sections 4 and 6.

5.1 Plain Text File

The wrapper for MPI_Finalize() writes the timing summaries in files called mpi_profile.taskid. The mpi_profile.0 file is special: it contains a timing summary from each task. Currently for scalability reason, by default only four ranks will generate plain text file: rank 0, rank with (min,med,max) MPI communication time. To change this default setting, please refer to the TRACE_ALL_RANKS environment variable.

An example of mpi_profile.0 file is shown as follows:

```
elapsed time from clock-cycles using freq = 700.0 MHz
-----------------------------------------------------------------
MPI Routine                 #calls     avg. bytes      time(sec)
```

```
---------------------------------------------------------------
MPI_Comm_size                1            0.0          0.000
MPI_Comm_rank                1            0.0          0.000
MPI_Isend                   21        99864.3          0.000
MPI_Irecv                   21        99864.3          0.000
MPI_Waitall                 21            0.0          0.014
MPI_Barrier                 47            0.0          0.000
---------------------------------------------------------------
total communication time = 0.015 seconds.
total elapsed time       = 4.039 seconds.

---------------------------------------------------------------
Message size distributions:

MPI_Isend                #calls    avg. bytes      time(sec)
                              3           2.3          0.000
                              1           8.0          0.000
                              1          16.0          0.000
                              1          32.0          0.000
                              1          64.0          0.000
                              1         128.0          0.000
                              1         256.0          0.000
                              1         512.0          0.000
                              1        1024.0          0.000
                              1        2048.0          0.000
                              1        4096.0          0.000
                              1        8192.0          0.000
                              1       16384.0          0.000
                              1       32768.0          0.000
                              1       65536.0          0.000
                              1      131072.0          0.000
                              1      262144.0          0.000
                              1      524288.0          0.000
                              1     1048576.0          0.000

MPI_Irecv                #calls    avg. bytes      time(sec)
                              3           2.3          0.000
                              1           8.0          0.000
                              1          16.0          0.000
                              1          32.0          0.000
                              1          64.0          0.000
                              1         128.0          0.000
                              1         256.0          0.000
                              1         512.0          0.000
                              1        1024.0          0.000
                              1        2048.0          0.000
                              1        4096.0          0.000
                              1        8192.0          0.000
                              1       16384.0          0.000
                              1       32768.0          0.000
                              1       65536.0          0.000
                              1      131072.0          0.000
                              1      262144.0          0.000
                              1      524288.0          0.000
                              1     1048576.0          0.000

---------------------------------------------------------------

Communication summary for all tasks:

  minimum communication time = 0.015 sec for task 0
  median  communication time = 4.039 sec for task 20
  maximum communication time = 4.039 sec for task 30

taskid  xcoord  ycoord  zcoord  procid   total_comm(sec)   avg_hops
     0       0       0       0       0            0.015       1.00
     1       1       0       0       0            4.039       1.00
     2       2       0       0       0            4.039       1.00
```

```
     3       3       0       0       0         4.039         4.00
     4       0       1       0       0         4.039         1.00
     5       1       1       0       0         4.039         1.00
     6       2       1       0       0         4.039         1.00
     7       3       1       0       0         4.039         4.00
     8       0       2       0       0         4.039         1.00
     9       1       2       0       0         4.039         1.00
    10       2       2       0       0         4.039         1.00
    11       3       2       0       0         4.039         4.00
    12       0       3       0       0         4.039         1.00
    13       1       3       0       0         4.039         1.00
    14       2       3       0       0         4.039         1.00
    15       3       3       0       0         4.039         7.00
    16       0       0       1       0         4.039         1.00
    17       1       0       1       0         4.039         1.00
    18       2       0       1       0         4.039         1.00
    19       3       0       1       0         4.039         4.00
    20       0       1       1       0         4.039         1.00
    21       1       1       1       0         4.039         1.00
    22       2       1       1       0         4.039         1.00
    23       3       1       1       0         4.039         4.00
    24       0       2       1       0         4.039         1.00
    25       1       2       1       0         4.039         1.00
    26       2       2       1       0         4.039         1.00
    27       3       2       1       0         4.039         4.00
    28       0       3       1       0         4.039         1.00
    29       1       3       1       0         4.039         1.00
    30       2       3       1       0         4.039         1.00
    31       3       3       1       0         4.039         7.00

MPI tasks sorted by communication time:
taskid  xcoord  ycoord  zcoord  procid    total_comm(sec)  avg_hops
     0       0       0       0       0           0.015         1.00
     9       1       2       0       0         4.039         1.00
    26       2       2       1       0         4.039         1.00
    10       2       2       0       0         4.039         1.00
     2       2       0       0       0         4.039         1.00
     1       1       0       0       0         4.039         1.00
    17       1       0       1       0         4.039         1.00
     5       1       1       0       0         4.039         1.00
    23       3       1       1       0         4.039         4.00
     4       0       1       0       0         4.039         1.00
    29       1       3       1       0         4.039         1.00
    21       1       1       1       0         4.039         1.00
    15       3       3       0       0         4.039         7.00
    19       3       0       1       0         4.039         4.00
    31       3       3       1       0         4.039         7.00
    20       0       1       1       0         4.039         1.00
     6       2       1       0       0         4.039         1.00
     7       3       1       0       0         4.039         4.00
     8       0       2       0       0         4.039         1.00
     3       3       0       0       0         4.039         4.00
    16       0       0       1       0         4.039         1.00
    11       3       2       0       0         4.039         4.00
    13       1       3       0       0         4.039         1.00
    14       2       3       0       0         4.039         1.00
    24       0       2       1       0         4.039         1.00
    27       3       2       1       0         4.039         4.00
    22       2       1       1       0         4.039         1.00
    25       1       2       1       0         4.039         1.00
    28       0       3       1       0         4.039         1.00
    12       0       3       0       0         4.039         1.00
    18       2       0       1       0         4.039         1.00
    30       2       3       1       0         4.039         1.00
```

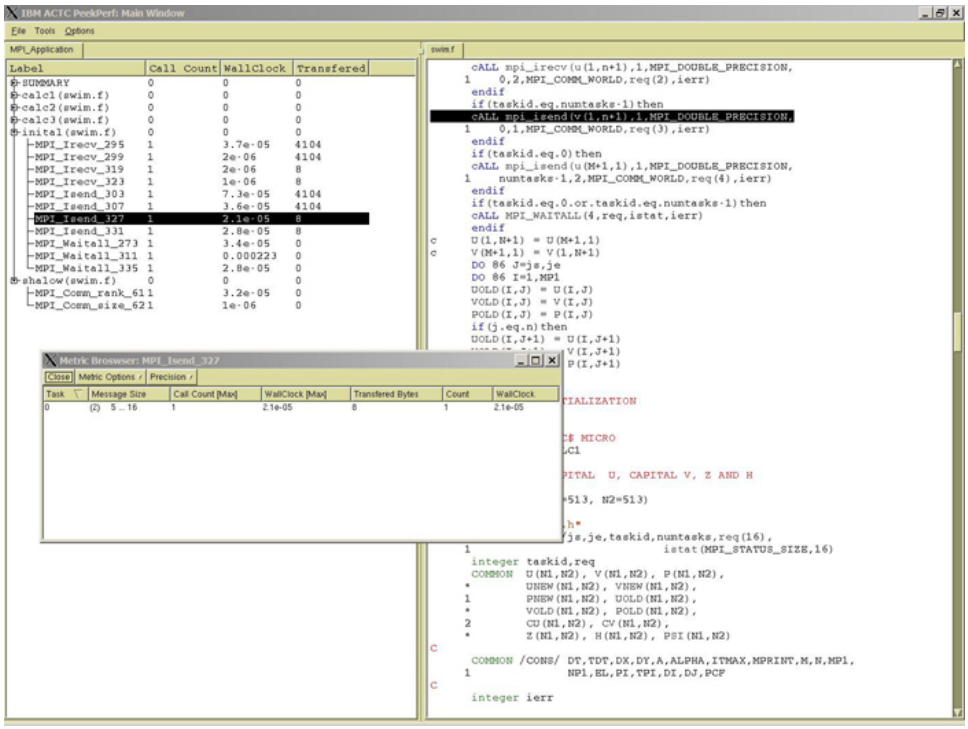

Figure 1: Peekperf

5.2 Viz File

In addition to the mpi_profile.taskid files, the library may also generate mpi_profile_taskid.viz XML format files that be viewed by using Peekperf as shown in Figure 1.

5.3 Trace File

The library will also generate a file called single_trace. The Peekview utility can be used (inside Peekperf or independently) to display this trace file as shown in Figure 2.

6 Configuration

In this section, we describe a more general way to make the tracing tool configurable, and thereafter allows users to focus on interesting performance points. By providing a flexible mechanism to control the events that are recorded, the library can remain useful for even very large-scale parallel applications.

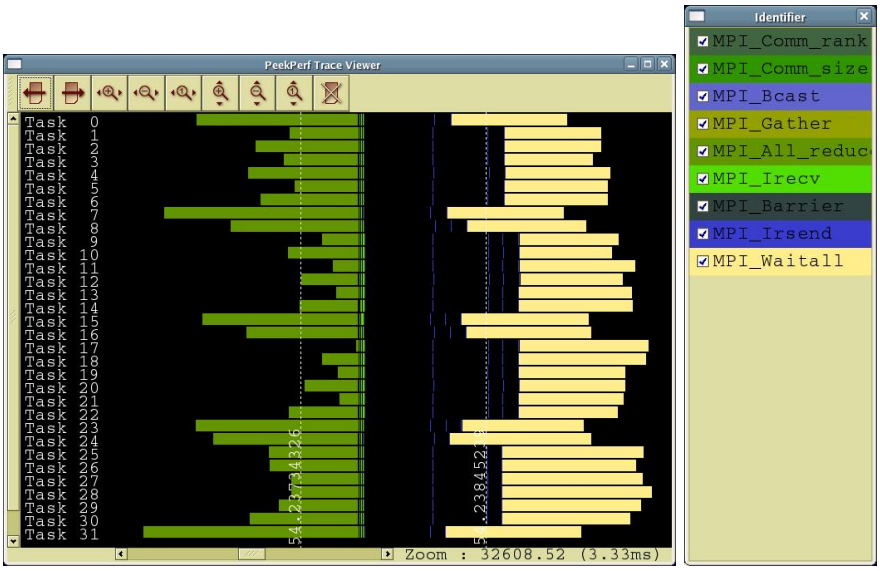

Figure 2: Peekview

6.1 Configuration Function

There are three functions that can be rewritten to configure the library. During the runtime, the return values of those three functions decide what performance information to be stored, which process (MPI rank) will output the performance information, and what performance information will be output to files.

- `int MT_trace_event(int);` Whenever a MPI function (that is profiled/traced) is called, this function will be invoked. The integer passed into this function is the ID number for the MPI function. The return value is 1 if the performance information should be stored in the buffer; otherwise 0.
- `int MT_output_trace(int);` This function is called once in the MPI_Finalize(). The integer passed into this function is the MPI rank. The return value is 1 if it will output performance information; otherwise 0.
- `int MT_output_text(void);` This function will be called inside the `MPI_Finalize()` once. The user can rewrite this function to customize the performance data output (e.g., user-defined performance metrics or data layout).

6.2 Data Structure

Each data structure described in this section is usually used with associated utility function described in Section 6.3 to provide user information when implementing the configuration functions described in Section 6.1.

- `MT_summarystruct`

  This data structure holds statistics results including MPI ranks and statistical values (e.g., Min, Max, Median, Average and Sum). The data structure is used together with MT_get_allresults() utility function.

```
struct MT_summarystruct {
        int min_rank;
        int max_rank;
        int med_rank;
        void *min_result;
        void *max_result;
        void *med_result;
        void *avg_result;
        void *sum_result;
        void *all_result;
        void *sorted_all_result;
        int *sorted_rank;
};
```

- `MT_envstruct`

  This data structure is used with MT_get_environment() utility function. It holds MPI process self information includes MPI rank (mpirank), total number of MPI tasks (ntasks), and total number of MPI function types (that are profiled/traced; nmpi). For Blue Gene/L, it also provides the process self environment information including x,y,z coordinates in the torus, dimension of the torus (xSize, ySize, zSize), the processor ID (procid) and the CPU clock frequency (clockHz).

```
struct MT_envstruct {
        int mpirank;
        int xCoord;
        int yCoord;
        int zCoord;
        int xSize;
        int ySize;
        int zSize;
        int procid;
        int ntasks;
        double clockHz;
        int nmpi;
};
```

- `MT_tracebufferstruct`

  This data structure is used together with MT_get_tracebufferinfo() utility function. It holds information about how many events are recorded (number_events) and information about memory space (in total/used/available; MBytes) for tracing.

```
struct MT_tracebufferstruct {
        int number_events;
        double total_buffer; /* in terms of MBytes */
        double used_buffer;
        double free_buffer;
};
```

- `MT_callerstruct`

  This data structure holds the caller's information for the MPI function. It is used with MT_get_callerinfo() utility function. The information includes source file path, source file name, function name and line number in the source file.

```
struct MT_callerstruct {
        char *filepath;
        char *filename;
        char *funcname;
        int lineno;
};
```

- `MT_memorystruct` (Blue Gene/L Only)

  Since the memory space per compute node on Blue Gene/L is limited. This data structure is used with MT_get_memoryinfo() utility function to provide memory usage information.

```
struct MT_memorystruct {
        unsigned int max_stack_address;
        unsigned int min_stack_address;
        unsigned int max_heap_address;
};
```

6.3 Utility Functions

- `long long MT_get_mpi_counts(int);`

  The integer passed in is the MPI ID and the number of call counts for this MPI function will be returned. The MPI ID can be one of IDs listed in Table 2.

- `double MT_get_mpi_bytes(int);`

  Similar to the MT_get_mpi_counts(), this function will return the accumulated size of data transferred by the MPI function.

- `double MT_get_mpi_time(int);`

  Similar to the MT_get_mpi_counts(), this function will return the accumulated time spent in the MPI function.

- `double MT_get_avg_hops(void);`

  If the distance between two processors p, q with physical coordinates (x_p, y_p, z_p) and (x_q, y_q, z_q) is calculated as $Hops(p,q) = |x_p - x_q| + |y_p - y_q| + |z_p - z_q|$. We measure the $AverageHops$ for all communications on a given processor as follows:

$$AverageHops = \frac{\sum_i Hops_i \times Bytes_i}{\sum_i Bytes_i}$$

COMM_SIZE_ID	COMM_RANK_ID	SEND_ID
SSEND_ID	RSEND_ID	BSEND_ID
ISEND_ID	ISSEND_ID	IRSEND_ID
IBSEND_ID	SEND_INIT_ID	SSEND_INIT_ID
RSEND_INIT_ID	BSEND_INIT_ID	RECV_INIT_ID
RECV_ID	IRECV_ID	SENDRECV_ID
SENDRECV_REPLACE_ID	BUFFER_ATTACH_ID	BUFFER_DETACH_ID
PROBE_ID	IPROBE_ID	TEST_ID
TESTANY_ID	TESTALL_ID	TESTSOME_ID
WAIT_ID	WAITANY_ID	WAITALL_ID
WAITSOME_ID	START_ID	STARTALL_ID
BCAST_ID	BARRIER_ID	GATHER_ID
GATHERV_ID	SCATTER_ID	SCATTERV_ID
SCAN_ID	ALLGATHER_ID	ALLGATHERV_ID
REDUCE_ID	ALLREDUCE_ID	REDUCE_SCATTER_ID
ALLTOALL_ID	ALLTOALLV_ID	

Table 2: MPI ID

where $Hops_i$ is the distance between the processors for ith MPI communication and $Bytes_i$ is the size of the data transferred in this communication. The logical concept behind this performance metric is to measure how far each byte has to travel for the communication (in average). If the communication processor pair is close to each other in the coordinate, the $AverageHops$ value will tend to be small.

- `double MT_get_time(void);`

  This function returns the time since MPI_Init() is called.

- `double MT_get_elapsed_time(void);`

  This function returns the time between MPI_Init() and MPI_Finalize() are called.

- `char *MT_get_mpi_name(int);`

  This function takes a MPI ID and returns its name in a string.

- `int MT_get_tracebufferinfo(struct MT_tracebufferstruct *);`

  This function returns the size of buffer used/free by the tracing/profiling tool at the moment.

- `unsigned long MT_get_calleraddress(int level);`

  This function will return the caller address in the memory.

- `int MT_get_callerinfo(unsigned long caller_memory_address, struct MT_callerstruct *);`

This function takes the caller memory address (from MT_get_calleraddress()) and returns detailed caller information including the path, the source file name, the function name and the line number of the caller in the source file.

- `void MT_get_environment(struct MT_envstruct *);`

  This function returns its self environment information including MPI rank, physical coordinates, dimension of the block, number of total tasks and CPU clock frequency.

- `int MT_get_allresults(int data_type, int mpi_id, struct MT_summarystruct *);`

  This function returns statistical results (e.g., min, max, median, average) on primitive performance data (e.g., call counts, size of data transferred, time...etc.) for specific or all MPI functions.

  The data_type can be one of the data type listed in Table 3 and mpi_id can be one of the MPI ID listed in Table 2 or ALLMPI_ID for all MPI functions.

COUNTS
BYTES
COMMUNICATIONTIME
STACK
HEAP
MAXSTACKFUNC
ELAPSEDTIME
AVGHOPS

Table 3: Data Type

- `int MT_get_memoryinfo(struct MT_memorystruct *);` (Blue Gene/L Only)

  This function returns the information for the memory usage on the compute node.

6.4 Example

In Figure 3 we re-write the MT_trace_event() and MT_output_trace() routines with about 50 lines of code (and use the default version of MT_output_text()) on Blue Gene/L. The function automatically detects the communication pattern and shuts off the recording of trace events after the first instance of the pattern. Also only MPI ranks less than 32 will output performance data at the end of program execution. As shown in the figure, utility functions such as MT_get_time() and MT_get_environment() help the user easily obtain information needed to configure the library. In this example, MT_get_time() returns the execution time spent so far and MT_get_environment() returns the process personality including its physical coordinates and MPI rank.

```
int MT_trace_event(int id) {

  ...
  now=MT_get_time();
  MT_get_environment(&env);
  ...

  /* get MPI function call distribution */
  current_event_count=MT_get_mpi_counts();

  /* compare MPI function call distribution */
  comparison_result
  =compare_dist(prev_event_count,current_event_count);

  prev_event_count=current_event_count;
  /* compare MPI function call distribution */
  if(comparison_result==1)
          return 0; /* stop tracing */
  else
          return 1; /* start tracing */
}

int MT_output_trace(int rank) {

  if (rank < 32)
          return 1; /* output performance data */
  else
          return 0; /* no output */
}
```

Figure 3: Sample Code for MPI Tracing Configuration

7 Final Note

7.1 Overhead

The library implements wrappers that use the MPI profiling interface, and have the form:

```
int MPI_Send(...) {
  start_timing();
  PMPI_Send(...);
  stop_timing();
  log_the_event();
}
```

When event tracing is enabled, the wrappers save a time-stamped record of every MPI call for graphical display. This adds some overhead, about 1-2 microseconds per call. The event-tracing method uses a small buffer in memory - up to 3×10^4 events per task - and so this is best suited for short-running applications, or time-stepping codes for just a few steps. To further trace/profile large scale application, configuration may be required to improve the scalability. Please refer Section 6 for details.

7.2 Multi-Threading

The current version is not thread-safe, so it should be used in single-threaded applications, or when only one thread makes MPI calls. The wrappers could be made thread-safe by adding mutex locks around updates of static data - which would add some additional overhead.

8 Contacts

- I-Hsin Chung (ihchung@us.ibm.com)
  Comments, corrections or technical issues.
- David Klepacki (klepacki@us.ibm.com)
  IBM High Performance Computing Toolkit licensing and distributions.